NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

SOFTWARE LIBRARY

A REUSABLE SOFTWARE ISSUE

by

Sherman G. Metcalf

June 1984

Thesis Advisor: Gordon H. Bradley

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Software Library - A Reusable Software Issue	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1984
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Sherman G. Metcalf	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943	12. REPORT DATE June 1984
	13. NUMBER OF PAGES 77
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Library; Program Library; Reusability; Generator

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents a conceptual view of a reusable "Software Library." Issues concerning the "software crisis" and its subsequent impact on software development are reviewed. The traditional library is described for the purpose of comparison with the Software Library. A particular example of the Software Library, the Program Library, is described as a prototype of a reusable library. A hierarchical structure for a Program Library is discussed as an approach to making the library entities easily

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102- LF- 014- 6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

accessible and retrievable. The role of application generators in the Program Library is described. The special features of Ada that support programming libraries are described. Finally, non code products in the Software Library are discussed.

Approved for public release; distribution unlimited.

Software Library - A Reusable Software Issue

by

Sherman G. Metcalf
Lieutenant, United States Navy
B.S., Purdue University, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

This thesis presents a conceptual view of a reusable "Software Library." Issues concerning the "software crisis" and its subsequent impact on software development are reviewed. The traditional library is described for the purpose of comparison with the Software Library. A particular example of the Software Library, the Program Library, is described as a prototype of a reusable library. A hierarchical structure for a Program Library is discussed and an approach to making the library entities easily accessible and retrievable. The role of application generators in the Program Library is described. The special features of Ada that support programming libraries are described. Finally, non code products in the Software Library are discussed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION AND BACKGROUND

The Department of Defense's (DoD) Annual Report FY '81, reported the DoD spent over $3 billion on software with these expenses being projected to grow to $30 billion per year by 1990 [Ref. 1]. These estimates are alarmingly high; what is perhaps worse is the projection that the costs of software maintenance will rise significantly above the cost of original development.

As this trend of increasing software costs continues, two questions come to mind: why are the costs factors so dramatic? and are the reasons resolvable with today's modern technology? The general response to the cause of the high cost of software usually centers on the highly publicized "software crisis." The crisis encompasses all software related problems, from the simpliest to the most complex. More specifically, when referring to software systems, the reasons for the crisis focus on the issues of the systems being nonresponsive to user needs, unreliable, excessively expensive, untimely, inflexible, difficult to maintain and not reusable [Ref. 2]. For the most part, these reasons establish the symptoms of the problem, rather than identify the problem itself. But, since the problem is not well defined, the solution may conceivably come through the alleviation of the symptoms.

To help solve portions of the software crisis, software tools and techniques must be developed. The development of products is but an initial step. The emphasis should be placed on the concepts associated with the software product. One of the more prevalent concepts to be addressed is that of software reusability. Because of its broad definition (as defined in a latter section), reusability closely

relates to other concepts like commonality, portability, modularity, maintainability and evolution. These relationships are described more precisely in [Ref. 3].

What makes reusability so crucial is the presumption that a well understood grasp of this concept could indeed resolve some of the acknowledged symptoms of the software crisis. To suggest that reusability alone could solve the crisis is ridiculous. To use the concept in conjunction with a proven software methodology would seem more realistic. But there is little evidence that any practical software development methodology along these lines will be available in the near future.

The Software Library has been proposed as a conceptual software product designed to help solve many of the software related problems. Before the Software Library can be introduced as a possible solution to any of the problems included in the software crisis, it must provide to the user the ability to identify and resolve the many related symptoms. The Software Library is not a new or modern concept. However, as proposed in this thesis it can be designed as a hierarchical library able to respond to some of the aforementioned symptoms. The extent to which this software product can resolve the bottleneck in software development is uncertain, but the potential does exist, as will be discussed.

A. SOFTWARE LIBRARIES

1. History of Program Libraries

The value of Software Libraries has been recognized since the introduction of computers and associated programs. In the early days of computers, libraries were mainly used

as repositories for commonly used software. It was not until the latter 1960's and early 70's, when the economic cost factors (i.e., production and maintenance) of software began to rise, that the significance of the Software Library became highly evident. There were other factors instrumental in reestablishing interest in libraries: increasingly complex problems, (e.g., mathematical), needless duplication of code and code which was usually less than reliable. Since a large number of the components to be housed in the library were mathematical in nature, it became necessary to produce a library that was reliable, general (able to service a broad group of users) and accurate.

To fulfill the requirements sought by the users of the libraries, the IMSL (International Mathematical and Statistical Libraries, Houston, Texas) and the NAG (Numerical Algorithms Group, Oxford, England) Libraries were introduced. From these libraries and others of this era, the concept of the Program Library evolved.

2. A General Definition of a Program Library

The IMSL and NAG Libraries can be considered as good Software Libraries, highly effective in accomplishing their design goals. That is not to say, that either would provide the best basis for defining the Program Library envisioned by this author. To give an appropriate definition, requirements concerning today's (1984) technology must be incorporated. One of the basic demands of current users of a library is for organized storage, search and retrieval of entities (e.g., programs and their components) within the library. Other concerns include the ability to manipulate (i.e., modify, link, update and list) entities. These issues are important because the users of a library will in

general be people other than the author of the entities. Finally, the library should have such assets as: speed, efficiency and effectiveness. The requirements mentioned above, lead to a definition of a Program Library: "A standardized collection of proven entities to be stored, retrieved and manipulated by a user."

3. <u>Status of Program Libraries</u>

A review of existing Program Libraries shows that there is wide variability in quality. According to [Ref. 4] the SHARE Program Library represents an unreliable source of software. Even though the library provides many useful and shareable routines, the number of routines which fail to work as advertized is unacceptably high. On the other extreme, IMSL provides a library that is highly successful: a programmer who has the resources of the IMSL library is literally wasting time and money if he or she resorts to writing software which performs any of the proven functions supplied by IMSL.

With the success of the IMSL and similar libraries, why has there not been more widespread use of the Program Library? Why is research on Program Libraries virtually nonexistent? The economics involved could be part of the reasoning or possibly there is a lack of understanding of what a truly quality library should offer a user.

4. <u>Evaluation of Program Libraries</u>

The significance of the Program Library has been emphasized over the past 20 or so years, but still there has not been signs of growth in the number of libraries. Those Program Libraries (or similar software tools) that have

proven to be efficient and cost effective have dominated the computer industry and have established guidelines for future developmental software. Rice and Schwetman suggest in [Ref. 4], that there are at least three requirements which should be present in any quality library:

1. A large supply of useful, reliable parts
2. A catalog of parts, making them easy to locate and evaluate, and
3. A mechanism for connecting parts together, so as to form more complex objects.

Using these requirements as an evaluation tool, an evaluation scheme, as shown in Table I, can be used to rate existing libraries. The requirements as enumerated above are not all inclusive and without economic justification (in time and money) the library can not be fully evaluated against these or any other requirements. With established methods of evaluating Program Libraries and economic reasons justifying future developments in this area; why has this not been practiced more widely? In seeking the answer, this thesis suggests that many of the motivation factors (i.e., reusability, portability, generality, etc.) have not been completely understood. Once these and other issues are incorporated into the evaluation scheme for quality Program Libraries, the motivation necessary to design these and other software products can be better understood. Some of these motivation issues will be explained in this thesis, hopefully, this will benefit the future developmental software products.

E. ECONOMICS OF THE SOFTWARE LIBRARY

It has been stated in [Ref. 5], that by 1990 there could be as many as 1.2 million programmers in demand with the

TABLE I

Evaluation of Program Libraries

TYPE LIBRARY	REQUIREMENTS	RATINGS*	REMARKS
IMSL	1	E	When used with mathematical and statistical applications
	2	A	Available, but not easy to locate routines
	3	P	Has no interconnection scheme, depends on outside programming
NAG	1	E	Primarily for mathematical and statistical applications
	2	E	Has its own indexing scheme, which is accessible, partially in machine readable form
	3	E	Has built-in linking mechanism
UNIX	1	E	Has large number of commands and programs for various applications
	2	A	Available in manuals and KWIC indices
	3	E	Uses pipe mechanism as an interconnection scheme, but only for single streams of characters

*Characteristic ratings as follows:
E - Excellent
A - Average
P - Poor

actual supply of capable programmers failing to rise fast enough to close the gap between supply and demand. While this situation only represents a small portion of the overall crisis, there would seem to be enough of a problem to warrant more concern than is presently exhibited. What

appears even more astonishing, is the fact that, these and other problems have not resulted in massive efforts to develop software products which could possibly resolve some of the problems of supply and demand.

As cited on numerous occasions, the Software Library has been around for a number of years, but yet it has not significantly evolved into the type of product capable of resolving any of the major effects of the software crisis. This could be due to the lack of Software Libraries in industry. So, why are they so scarce? It could be that the concept of a Software Library, more specifically a Program Library, fails to project substantial savings (i.e., in time, in money or in productivity) to the user. It could also be that each organization is waiting for the other to take the first step. Of course, it could be due to something other than any issue discussed thus far. To pursue the question even further, one must wonder if the concept of a reusable Program Library will actually reduce the amount of redundancy in program writing. Or will the time spent searching for existing library components, outweigh any savings? These are but a few of the issues that may have lessened interest in evolving the Software Library. Although the economic pitfalls of a software product, in this case the Software Library, may never be fully realized or resolved, it is still the responsibility of the designer, the implementor and the user to insure that the many questions surrounding the economic issues have been addressed.

Once there is a better understanding of reusability and the Software Library, there can be a more widespread use of the concepts. That is to say that certain issues such as time spent reproducing and testing a program can be better utilized. There are other economic issues each affecting the software crisis in some unique manner. The economic

issues are important to the future of software development. An understanding of the problem is not enough to solve the problem, but with the implementation of such concepts as reusability, the software crisis may be reduced to a more manageable problem.

1. Reusability and Portability

Of the many motivations, driving the need for a Software Library, there are two very closely related notions. They are portability and reusability. In seeking a relationship which best exemplifies the closeness between the two concepts, the following representation seems appropriate: reusability should be considered a necessary, but not sufficient condition for portability. This shows the relative importance of reusability, however each concept will be discussed.

a. Reusability

Reusability has been identified as a key to the effectiveness of the Software Library and as a concept for helping to solve the previously mentioned software crisis. Unfortunately, there is not one unique definition to support the concept of a Software Library associated with reusable software issues. Therefore to establish a basis for understanding, the following definition will be used: "Software resources of a capital nature which are used in the development or maintenance of software products with end uses different from that of the component resources." Further clarification also provided by the reference encompasses any information generated at any time throughout the software life-cycle. Also, a component resource is described as a modular product of the software life-cycle, possessing the characteristics of being highly cohesive [Ref. 6].

In [Ref. 6], the author presents a list of major factors which dictate the usefulness of reusable software. The greatest concern is that acceptance of a product would not be forthcoming if the product is not understood. Thus, if a product does not appear easy to use and economically feasible, then there would be little desire to understand it. A product can not prove itself, if it is not used.

Since the concept of "reusing" software has been around for so long, technological improvements in this field should be researched. The conceptual Program Library represents a source to be used as a guide for future development of reusable software.

b. Portability

The concept of portability has existed since the discovery that large savings can be realized from the distribution of good software. But, as touched on by John Rice [Ref. 7], the dissemination of quality software is opposed by formidable barriers, such as the dependency of software on machines and the idiosyncrasies of compilers and operating systems. Even though Rice was referring specifically to numerical computation software, his comments warrant consideration by any organization contemplating the development and transportability of a new software product.

Portability deals with the designing of a product that will minimize the amount of change required to move the product from one environment to another. Portability also takes into consideration most issues of compatibility which affect the transportability of a software product.

The Program Library, while not specifically designed as a portable software product, should have capabilities consistent with portability issues. The reasoning is that portability issues represent a form of enticement to the user. After being influenced to use a product its benefits can be better understood. Thus, the added facility of portability can be treated as a motivational concept for helping the conceptual Program Library resolve some of the problems inherent to the software crisis.

The environment of the Program Library and the user should determine which concepts require the most emphasis. In an attempt to reduce the effects of the software crisis both concepts (portability and reusability) should be considered essential to the user of the Program Library.

2. Standardization of the Software Library

The efficient and effective understanding of software products written by others is one of the critical problems in software development. Much of the labor expense in software development is involved in the understanding of the various software products. One approach to this problem has been to apply standards to software products. Standardization allows people who are familiar with parts of a software product to more easily become familiar with other parts. Some of the areas of concern to standardization include:

Format
Documentation
Specification
Test Plans
Error Handling
Modularity

The standardization of products makes it faster (and thus more efficient) to understand a software product one has not seen before. Standardization is critical to the Software Library:

1) if users other than the originator are to easily access and retrieve items in the Library,
2) if items in the library are to be incorporated without change into other larger standardized software products,
3) and if the library is to be built and maintained efficiently.

Since standardization represents such a critical aspect in software development there should be management mechanisms established to enforce standards. These mechanisms should not discourage the use of the library, instead they should suggest an ease of use preferable to writing one's own code.

3. Reliability in a Software Library

"The ever-increasing expectations and needs of large organizations and the advent of large, cheap memories has led to the creation of ever- larger information systems. One of the results has been the discovery that while a small system could often be thoroughly tested, for all practical purposes large systems of interacting hardware, software and people could be rendered useless because of unreliability. Since the physical and economic consequences of information systems failure may be very great, interest in reliability has grown also," [Ref. 8].

One definition of reliability found in [Ref. 8] suggests the following: "a piece of software that is correct with respect to stated requirements and that, further, is

able to withstand unanticipated demands as well." Demands for reliability date back a number of years to when usage of the Software Library began. The primary concerns then were, that the library's reliability be exhibited, in its accuracy and in its mathematical stability. The need for reliability in a Software Library has not changed over time, but there is little evidence that software development has met these demands with more reliable Software Libraries. The financial investments and research in software development seems to grow slowly, even though the reasons justifying such an investment seem overwhelming.

With a renewed belief that there is a need for such reliable software products, all that will be required is a product that will suggest economic reasons for industry and DoD to invest in further research. The conceptual Software Library proposed by this thesis will hopefully influence such interest.

4. <u>Generality</u>

For a Software Library to support the concept of reusability, its design and the design of the components within its structure, must offer a certain amount of generality. Parnas [Ref. 9] states that software can be considered "general" if it can be used without change in a variety of situations. Thus, the concept of reusability which emphasizes modification (i.e., change) represents a conflict with the concept of generality. Parnas also states that software can be considered "flexible" if it is easily changed to be used in a variety of situations. This notion of flexibility is more consistent with reusability.

Based on Parnas' definitions the best way of achieving generality in a proposed reusable product is to

have some form of balance between the concepts of generality and flexibility. The actual balance is between the run-time costs to be paid for generality and the design-cost inherent to flexibility. The designer of a software product may not readily find this balance. But if he or she makes a conscientious effort at deciding this issue, a resulting reusable product will be more achieveable.

C. GENERAL DEFINITION OF A SOFTWARE LIBRARY

For the most part the Software Library and the issues surrounding it have stressed code oriented goals. While the Software Library is designed to support various forms of code, to center on this aspect is not consistent with the expectations of the overall software product. The Software Library will serve the user and his organization best if it is defined in a broader context. The first step is to insure that the semantics of the term "entity" include documentation, specifications, designs, requirements and test plans, as well as code.

The more general definition of a Software Library is "a standardized collection of reusable software products designed to enhance economic savings through the manipulation and modification of its reusable entities."

D. STRUCTURE OF THE THESIS

Chapter II discusses the automated traditional library. Since the user's requirements for a traditional library are similar to those for a Software Library, this chapter gives some insight into the functions of the conceptual Program Library. Chapter III presents criteria to assist in

recognizing quality Software Libraries. It compares various existing Software Libraries and suggests how they might be used to establish guidelines for future developmental libraries, specifically the Program Library.

Chapter IV introduces a hierarchical representation of a Program Library that is unlike most contemporary Program Libraries. The discussion stresses how this structure can improve the library's operation with regard to software reusability. Chapter V describes a application generator. Its design consists of program generators structured in a hierarchical fashion at a level higher than the highest level in the Program library. This chapter will explain how this software product will assist the user.

Chapter VI outlines an on-line method of searching and retrieving entities in the Program Library. The Library Reference Guide discussed in this chapter represents a manageable interface between the library and the user. In Chapter VII, the programming language Ada is provided as an existing language capable of meeting some of the requirements of the conceptual Program Library. Many of the concepts in Ada are still being researched, but in general Ada is a language with potential usefulness for a Program library.

Chapter VIII discusses how the concept of a Software library can be extended to non code software products. These products include: documentation, requirements, specifications, designs and test plans. Chapter IX is the concluding chapter. It presents a general overview of the thesis.

II. THE AUTOMATION OF THE TRADITIONAL LIBRARY

The traditional library represents a wealth of knowledge in the form of books, journals, serials, reports and so forth. Therefore the concept behind the automation of such massive resources presents the stiffest of challenges to modern technology. The complexity of the challenge is increased because of the usual opposition to the changing of a so called "working system." To address this resistance to change, an aspect of automation beneficial to librarians and library users will be stressed.

The aspect of interest is the application of computers to information processing. A specific concern, familiar to the Software Library, is how to process the data needed for control over and for access to information. Another concern is in the approach used by an individual to interact with the computer system. Existing and future technology accompanied by conventional practices within the library should produce products able to respond to these and other concerns. These issues are resolvable given an adequate understanding of the distinction between requirements and the actual design. The gist of the distinction is that requirements are independent of any specific design for implementation. To convince the skeptics of the future of automation within the library, the basic criteria associated with existing library services should be discussed. The traditional library, as it stands, may not provide all the services expected of an automated system, therefore to view the library in the correct perspective, issues other than speed and efficiency should be introduced. Prior to discussing futuristic criteria for an automated system, the

expected functions of the library, as viewed by the user, must be described.

1. Requirements of the Automated traditional library

To the average user, the Automated traditional library is somewhat of a remote concept. Thus, to lessen this sense of remoteness, the knowledge of both the librarian (since he or she is in constant contact with the user) and the engineer (who has designed many automated systems) is required. The purpose is to combine this knowledge into a concerted effort for the design and implementation of the most effective user-friendly system. Based on research of user needs and on the interests of the user, there should be some form of communication network tying the user to an automated catalog and other bibliographic tools related to a large library or a system of libraries. Once a text is identified there should be quick delivery capability. There should definitely be some form of user interaction with the system, thus providing responsive query services to the user while he or she attempts to make series of rapid and repeated searches. Ease of access to the information must be provided by terminals (local and remote). Finally the system should display detailed information of a text and prior to responding to a request for a hard copy the system should provide to the user the ability to view pages of selected works. An important point to be stressed is that the functions described above are not to be thought of as independent functions, instead certain, if not all, are interrelated.

With the requirements of the automated system as suggested above, the selection of performance criteria from the users point of view can now be presented in the next

sections. It must be reemphasized that these criteria are only to be looked upon as guidance towards fulfilling the suggested requirement, and as requirements change so do the performance criteria (this suggests the need for flexibility in design).

2. Associated Performance Criteria

The performance criteria which are suggested as being necessary to the user, include the following:

1) user interaction with computer
2) aids to browsing textual information
3) a user-indexed library
4) access to different levels of information
5) communication between remote sources
6) extensive software tools
7) rapid response time

Although each of the aforementioned criteria is a major concern to the user, it is not within the scope of this thesis to describe in detail each criteria. So as to remain consistent with the overall purpose (i.e., the discussion of the conceptual Program Library), only the performance criteria associated with the user interaction with the computer will be discussed in any detail.

The interaction required between the user and the Automated traditional library, should not be thought of as removed from the control of the librarian. That is to say, the librarian is an integral part of the automated system. To be more specific, the librarian exists as a reference source capable of providing expert reference assistance in specific disciplines where detailed knowledge is required. He or she would also be expected to have access to other

librarians, thus increasing the degree of detail available on a given subject. The triangle created between the user, the librarian and the automated system add emphasis to the need for an effective communication network, and therefore, the need for a user/librarian interaction with the computer becomes more essential.

Present day technology suggests that the terminal keyboard is the most adequate form of interaction between the automated library system and the user. In keeping with the notion of simplicity, only a limited number of terminal related functions will be identified. Don R. Swanson [Ref. 10], a librarian with aspirations for designing a mechanized library, presented his concept of interaction under the definition of "programmed interrogation." In his presentation of the term programmed interrogation, he suggests six major "process control" keys used to provide the user with an initial set of choices at a console. These six keys are consistent with the terminal related functions suggested by this thesis. Therefore, the functions presented will be briefly described with Swanson's concepts in mind.

The first function necessary for a good working environment is labeled "specific work." Its purpose is to identify the request for a specific book, journal or report by means of author, title, publisher, or other descriptive (non-subject) information.

The next function labeled "subject selection" permits retrieval of material based on subject classification, index or keywords. It also allows retrieval of specific information and finally it permits browsing of the above information.

Another function labeled "previous selection" allows the user the ability to select any material he or any other specified person has used before.

The "similarity selection" is another function used to initiate a chain of bibliographic citations that satisfy specified work.

The function labeled "combination" allows the linking of any two functions.

The next function is labeled "sequence display" for its ability to step the display from one display to another.

A final function labeled "microfilm view" is used to call for a microfilm display of selected portions of any work identified on the CRT display.

The function labels, as described above, are not designed to suggest an all inclusive view of the terminal keyboard necessary to provide user interaction with the system. But, what it does suggest is a selection of functions considered basic to the operation of the library. Once the inquiry-response interaction has been effected between the user and the automated system, a basic format (possibly based on the bibliography) can be established as a guide or training device in the use of the system. With this guide, the user has an example of the response received from a properly formulated inquiry. Issues in regards to whether an inquiry is too broad, too narrow or too ambiguous should become more obvious as the interaction become more frequent. The underlying result is that, the user improves on his or her level of understanding. The Automated traditional library once understood, could be used effectively as a tool for increasing the research potential of the user and of the librarian.

III. CHARACTERISTICS RELATIVE TO A PROGRAM LIBRARY

A. EXISTING CHARACTERISTICS

Two organizations have produced large, portable, good quality and inexpensive libraries. They are IMSL (International Mathematical and Statistical Libraries, Houston, Texas) and NAG (Numerical Algorithms Group, Oxford, England). Both libraries are evaluated with regard to their existing characteristics and the characteristic primarily suited to the reusability concept. Although the software developed by these two groups consists largely of numerical subroutine, this does not exclude the feasibility of using their concepts on other types of software products (i.e., non-numerical).

In discussing the libraries developed by IMSL and NAG, the author is not implying that the characteristics represented by each is better or worse than any other. But the objectives of the two libraries are close to those desired in the conceptual Program Library. The characteristics or lack of will be discussed for both the IMSL and NAG libraries and hopefully, the concept of the Program Library will become evident to the user.

B. THE IMSL LIBRARY

The IMSL library consists of over 400 high quality mathematical and statistical subroutines. These subroutines represent programs derived from a variety of sources (including ones written by IMSL). Regardless of the source, all programs are rewritten with a uniform (i.e., standard)

style. According to Rice [Ref. 7], quality control is exercised by:

(a) choosing good sources (the advisors, a board of 12-15 experts, assist in this regard)
(b) using knowledgeable programmers with good supervision (some of the senior IMSL people work regularly on the library programs)
(c) testing (reasonably exhaustive for new programs, check point testing for maintenance or new machine versions)
(d) continual upgrading

As proposed in [Ref. 11], the IMSL library has moved to a Fortran converter system where a master file contains all the information needed for each machine version of a program. Much of the standard information is not explicitly in the file. A converter program then automatically produces the program for a particular target machine. The master file is itself a Fortran program that runs on one of the machines. Thus portability is an attribute of the IMSL library.

The characteristics of the IMSL library subroutines and documentation are of major concern to a user. Aside from the standardization of the documentation, there should be a good understanding of the general attributes residing in the library. The attributes [Ref. 12] are as follows:

(a) Testing of the library subroutines were performed at several levels in various computer/compiler environments.
(b) For each routine which has some error detecting capabilities, the user is protected by default. That is if the user chooses to ignore error possibilities a warning, in the form of a printed message, is issued.
(c) Each routine conforms to established conventions in coding and documentation.
(d) Each routine was designed and documented to be used by technical personnel in fields of science, engineering, medicine, agriculture, . . . , and in research activities.
(e) Accuracy of results, clarity of documentation, and efficiency of coding were given first priority in development.
(f) Periodicals and books are referenced for users

interested in details of algorithm development.

(g) Often, tests for applicability of the algorithm are applied; the user is warned if the algorithm fails. Pitfalls to be avoided in usage are noted.

(h) All information pertaining to usage of one routine is in one place. Documentation is a configuration of typed material and computer readable documentation (in the form of comment lines).

(i) Computer readable documentation permits on-line access to basic documentation. Computer readable material is distributed with source code.

(j) All routines have documented examples of input and results.

(k) Designers and programmers (or IMSL personnel responsible for a code) are available to answer user questions.

The general attributes as mentioned are not all inclusive, but are enough to provide some understanding of the capacity of the library. To reinforce the integrity of the library, IMSL, as the sole source of any technical information regarding these routines, assumes total responsibility for the operation of any routine. To facilitate the retrieval of the various routines and their associated documentation, IMSL has established a directory of routines in which each routine has been placed in an alphabetized category. IMSL also provide a key-word-in-context (KWIC) listing which offers the user a quick reference to a routine given the user has knowledge of the title. This is not always beneficial since there are many cases where the title does not accurately reflect the contents of the routine. However, the concept of a key word retrieval mechanism must not be overlooked.

Although the IMSL library has the many characteristics mentioned above, the retrieval and manipulation of the routines is generally hidden from the user. Should the user desire access to the IMSL Library, the capability does exist and the routines can be incorporated into the user's program. There are problems encountered when attempting to interleave a user's program with the IMSL Library; usually

these problems are more evident in the production environment than in an institutional organization. The reason is the increased productivity expected by most production organizations. The IMSL Library can be used as a guide to conceptualize a functional Program Library with some extensions to its existing characteristics.

C. THE NAG LIBRARY

The NAG Library represents a high quality numerical algorithm library for general use by universities. It also, by design, represents a portable system. The NAG Library [Ref. 13] operates as follows:

> Programs are obtained from a contributor (usually an expert from one of the cooperating universities or research establishments) who chooses the method and then writes, tests and documents the program. The program is then given to a validator who is also an expert in the relevant area. He is to critically examine the merit of the algorithm and test the usability of the program and its documentation. Once a program is validated for general merit, it is then validated by the NAG Central Office in Office in regards to formatting, language standards, etc. Various software aids are used for this second stage of validation.

The NAG uses a master library file system (similar to the IMSL master file) which contains all versions of each program. It also keeps a complete history of the versions of each program. Due to the high level language (i.e., subsets of Fortran, Algol 60 and Algol 68) and machine parameterization, new machine implementations are essentially automatic (i.e., transparent to the user). When an implementation is accepted, the programs are returned to the NAG Central Office for inclusion in the master library.

There are stringent test programs for each library routine to assure equivalent performance of the NAG Library versions [Ref. 14]. According to [Ref. 15], the library history information and test programs in the master file have been found useful in developing a more portable library.

The NAG Library, in a similar manner to that of the IMSL Library, provides a working understanding of a subroutine library. With the concept of the NAG Library being used as a guide, the task of establishing a Program Library should seem obtainable.

D. OVERVIEW OF CHARACTERISTICS

While the IMSL and NAG libraries appear to set the guidelines for an effective Program Library, neither has the characteristics expected of a functionally reusable Program Library as proposed by this thesis. Specifically, both libraries have beneficial characteristics, but each neglects the issues of reusability (e.g., cataloging, key-word indexing and retrieval, etc.).

The characteristics of the IMSL and NAG Libraries which support the concept of the Program Library will be discussed and presented as feasible qualities to be associated with a good library. To broaden the perspective of a library, the characteristics and attributes of the IMSL and the NAG Libraries should be slightly modified and in some cases changed to fit new goals.

A closer look at the two libraries reveal the following goals for a possible Program Library:

(1) The design and implementation of the Program Library should be under the auspices of a group of experts from a wide range of sources (i.e., designers, programmers, etc.).

(2) The environment of the Program Library must be established and all testing must be accomplished within it.

(3) Each entity within the library should be considered for error detection requirements. Appropriate error handling capabilities must be outlined.

(4) Standardization of coding and documentation is mandatory, for all entities within the library or evolved from the library.

(5) The clientele or users of the Program Library should be identified and the library must support them.

(6) The developmental priorities should be set, so that any latter tradeoffs will be on minor details as opposed to major issues.

(7) The development of the entities within the library is important and although the actual specifics can not be placed in the library, references providing knowledge of the details should be made accessible to the user.

(8) The library should represent a user-friendly project. Thus when manipulating entities in the library there should be appropriate tests for applicability to the user's requirements, thereby making it possible to quickly identify and avoid some of the problems of parameterization.

(9) The library is to be its own best source of information. Any inquiries as to the use of the library will be answered by its own documentation, alleviating the need for exterior (i.e., books) information. This implies on-line access to both the documentation and the other entities as they are used in the library.

(10) For the new user of the library, there should be example inputs, results and formatting restrictions and guidelines.

(11) The organization responsible for the concept of the Program Library and its design and implementation should also be responsible to the users for continual updating and maintenance.

The aforementioned characteristics will provide a good quality library but what is lacking is the characteristics that will make the library reusable to the user and his or her organization. Suggested additional characteristics should include but not be limited to:

(1) The ability to select the most optimal entities, for the accomplishment of the user's task.

(2) Library browsing capability, prior to the selection of a component within.

(3) The ability to locate a component of a similar component with the use of key-words, indexing and cataloging.

(4) Manipulation and retrieval capabilities on entities once located.

(5) The ability to modify and combine authorized modules so as to possibly create larger modules in a hierarchical manner. This should be accomplished while keeping the parameter passing process transparent to the user.

E. SUMMARY

As a final comment on the IMSL and NAG Libraries, certain observations seem evident. One is that, as high a quality as the two libraries appear to be, there seems to be little to indicate that the issue of reusability is of any concern. This thesis does not deny that a good library could very easily be created from the image of either the IMSL or the NAG Library, however it could be said, that the

Program Library is a "Superset" (containing the combined characteristics) of the two libraries. The main intent is to provide characteristics for a reusable Program Library, with the belief that reusability produces increased productivity.

IV. THE PROGRAM LIBRARY

The Program Library represents a conceptual design responsive to the widest range of users (from the novice to the expert). The library is to be established around goals consistent with user's needs. To understand the conceptual design and implementation, the goals of the Program Library will be identified and explained.

A. GOALS OF A PROGRAM LIBRARY

Initially, the Program Library should be designed so as to be of benefit to a wide range of users. Included in this design should be considerations for reliability, modifiability, understandability, testability and efficiency.

Another goal is to have a library that has powerful capabilities and has the flexibility to be easily modified to fit specific user needs. With the design being centered on these issues, the potential to create useful libraries can be enhanced. This issue will be discussed in more detail in the following sections.

Another goal is to emphasize portability. Portability should stress the minimization of change as a software product is moved from one environment to another. Thus, portability in the Program Library will require moving from one environment to another, causing concerns over compatibility and parameter passing issues. These are some of the issues that should be dealt with by the designer of the product and recognized by the user. The concept of portability as a goal for the Program Library changes as the type of environment varies. That is, the concerns involved

in moving between two unrelated computer environments (i.e., IBM 360/370 and Cyber 205) are separate from those encountered in moving between environments located within a larger environment (e.g., the UNIX and VMS operating systems within the VAX computer system).

Finally and most importantly, the issue of reusability must be addressed and the concept incorporated into the library (from the design stage to the users application). The reusability issue should be a basis for the design, implementation and use of the Program Library. The concept behind reusability should not be limited to the design phase, since the extension of the concept down to the most primitive entities in the library will also enhance the user's programming task. Most users of a software product are interested in increased savings (in time and money) and increased productivity in programming. Reusability is a suggested path to these goals, and if the Program Library and its associated entities are to reach the desired goals, the concept of reusability must be implemented.

The goals cited above for the Program Library are by no means conclusive. They merely provide a conceptual overview of what a user should expect from an operational Program Library.

E. A HIERARCHICAL VIEW OF THE PROGRAM LIBRARY

The Program Library can be described in a hierarchical fashion. The hierarchy of the Program Library consists of entities embedded in multiple conceptual layers, each layer representing a library. An example of a hierarchically layered model of the Program Library is represented in Figure 4.1. The layers of the library represent three

conceptual levels which make-up the Program Library. In this three level example, the levels are classified as a Low Level Library (LLL), a Mid Level Library (MLL) and a High Level Library (HLL). Each level can then be described by how its associated routines are manipulated (i.e., reused, modified, etc.).

1. The Low Level Library

A routine or any entity at the lowest level is written in source code. It is a stand-alone entity or a routine which calls no other routine. Calls made to routines at this level are not exclusive to any one routine or package at a higher level. In fact, each routine at the higher level has access to each and every routine at a lower level. This does not preclude the ability to modify these routines or manipulate them as reusable software. But there is a implicit limitation on the size (i.e., number of lines of code) of the routines at this level. At this level a routine should be expected to handle only one action, thus giving validity to the term "single action routine."

2. The Mid Level Library

Each entity at this level is constructed of source language code derived from the linking (via subroutine calls) to lower level routines. Thus, the lower levels can be viewed as providing operations not available in existing (i.e., Mid level) code. At this level the size of the entities is of major importance to the capability of the library. This is evident in the fact that, even though the size of entities at this level is comparable (not necessarily larger) to the size of entities at the lower level, they are more capable because of the availability of calls

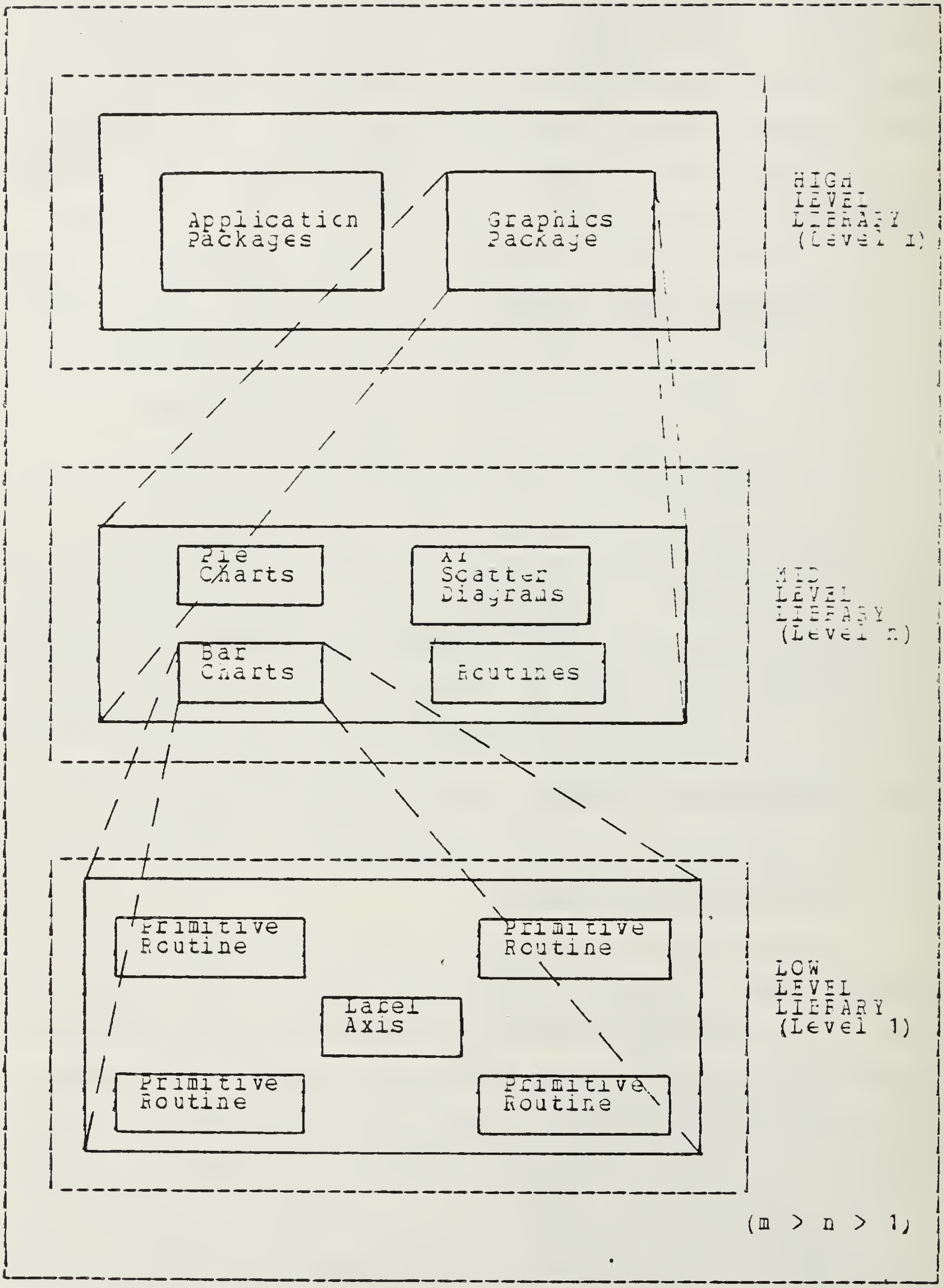

Figure 4.1 Hierarchy of the Program Library.

to lower levels. Unfortunately, at this and higher levels, as the capability of the library improves, the flexibility suffers. This is the fundamental tradeoff between levels of a Program Library; the increased power of higher level entities is available only by decreasing flexibility.

3. <u>The High Level Library</u>

The average user, not wanting to waste time writing a program from scratch, seeks code representative of complete components (i.e., application packages). To satisfy this request the Program Library has a high level library accessible by the user. As with the lower level libraries, its contents are still essentially modifiable and reusable. The size of the application packages are hopefully small (relative to packages constructed from lower-level libraries), but have significant capability. This again poses the issue of a tradeoff between capability and flexibility with the user being the beneficiary of the final result.

C. ADVANTAGES OF A HIERARCHICAL PROGRAM LIBRARY

In the structure shown in Figure 4.2, if a entity at the highest level (level 3) wants to make use of a entity at the lowest level (level 1), the calling sequence must make use of the entities at the mid level (level 2). And should the routine labeled B wish to use the routine labeled G, it must pass through the routines labeled A and C, since they are in the hierarchical calling sequence. This type design is similar to the designs that use the concept of stepwise refinement. Therefore the flexibility available to the user is limited.

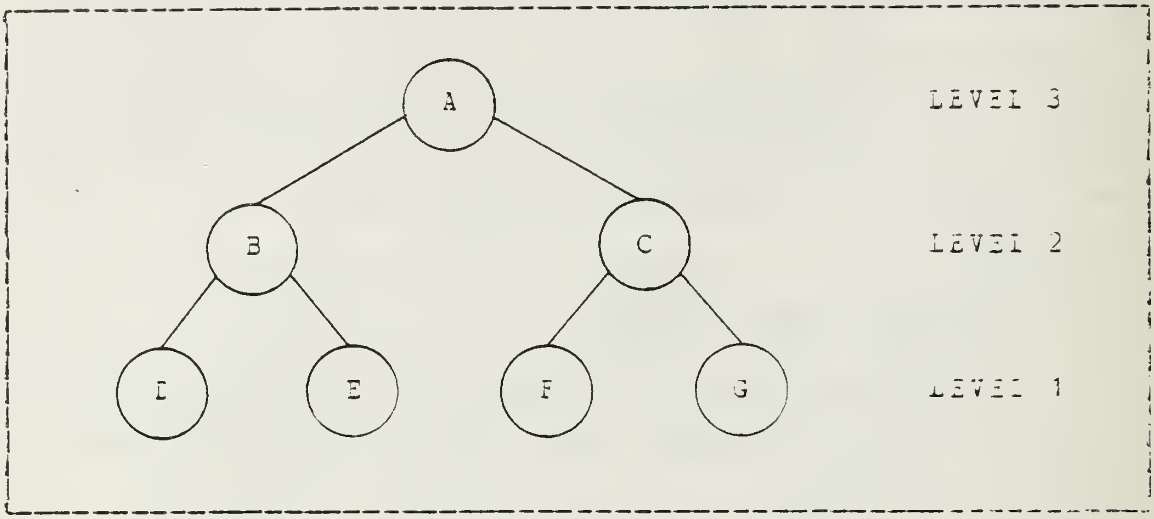

Figure 4.2 A Hierarchical Structure.

The Program library's structure as shown in Figure 4.3, offers most of the relations sought in Figure 4.2, but the unique distinguishing feature is the potential to deviate from the normal calling sequence. That is, the principles associated with stepwise refinement in a hierarchical structure are still relevant with this design. Though now, the user has the ability to perform manipulations (i.e., calls to routines) from high levels to low levels without passing through the middle levels. For an example, routine A at level 3 can make use of routines D, E, F, or G at level 1. Another example, illustrated in the figure, provides the ability for two or more routines at the same level (e.g., B and C of level 2) to make use of any routine at level 1 (e.g., D, E, F and G). A more indepth explanation of this and the previous mentioned relation can be found in the article by Parnas [Ref. 9] on the "uses" relation.

Even though the programs in the 'multiple level' Program Library may be identical to those in a single level design (similar to that of the IMSL Library), giving the user quick

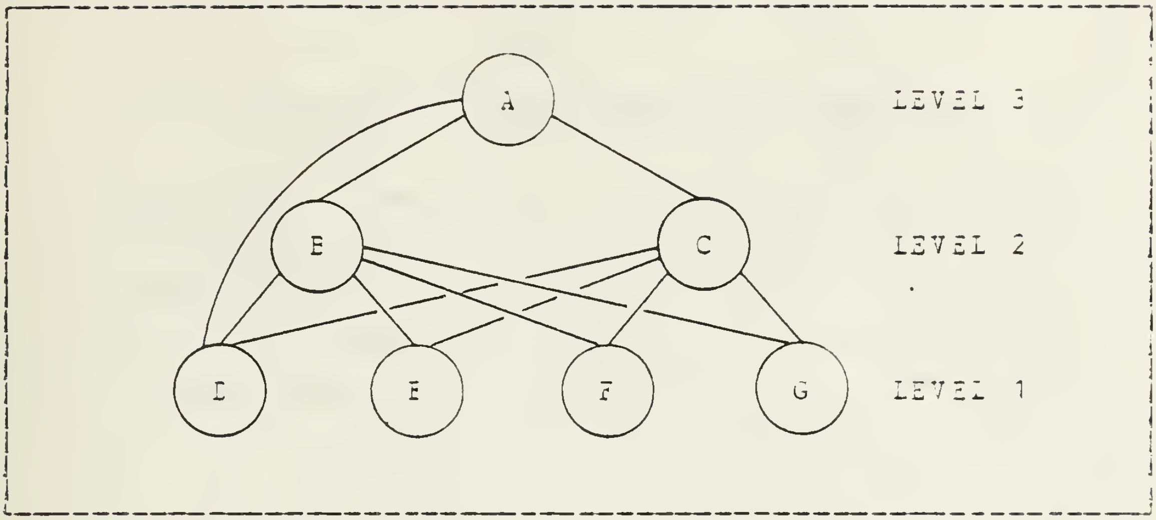

Figure 4.3 Hierarchical Structure of a Program Library.

access to the programs at the lower levels will allow him or her to use the library more effectively. With the hierarchical design, the user now has a large selection of routines for writing programs or high level applications. If there is a need to modify a high level program, the user only need to optimize the calling sequence, since the programs are written in terms of calls to lower level programs. Since the programmer can modify the program, he or she has the ability to add or delete the capability. Also with the use of calls to lower level programs the size of the higher level programs are not nearly as large as those used in different designs. Finally, the hierarchical design is consistent with the state-of-the-art technology, known as "modularity."

1. Power and Flexibility in a Program Library

Both power and flexibility of a Software Product begin in the design phase of its life-cycle and both affect the user's programming efficiency. The granularity (i.e.,

size) of the Program Library should represent a major ingredient in the library's bid for power and flexibility. In an attempt to provide both conceptual goals, while keeping the granularity of the library's entities as small as possible, any and all tradeoffs should be examined. One anticipated example of a tradeoff is due to the issues around entity size. That is, in order to maintain the size of entities, a hierarchical approach to program creation and modification should be used. As the entities are raised to a higher level in the hierarchy, the more capable the library will become, while the degree of flexibility is lessened. Although, the two concepts are equally important to the design and eventual implementation of the library, there will be instances where one will be prefered to the other. The designer and user of the Program Library should, at all times, seek an equitable balance between power and flexibility.

From the prospective of the users of the Program Library, programmers of any level of proficiency will be able to write applications easily. The novice should be able to implement entire applications with a minimum number of calls to lower level routines. A more experienced programmer should be able to generate a greater repertoire of routines for establishing applications.

V. THE ADDITION OF AN APPLICATION GENERATOR

The program Library has been described as a Software Product designed as a hierarchical structure of libraries. The concept provides the designer and the user a highly effective and reusable software tool. Even though the Program Library suggests to the user a new and "easy" method for the improvement to program productivity, it still requires that the user have some formal programming knowledge. Thus, it is not as "user-friendly" as expected from a product based on a high level language. A high level programming language that could be used to easily and succinctly express problems would be a very valuable tool for improving programmer productivity. One approach to this has been to investigate "Automatic Programming" systems. Balzer [Ref. 16], gives an example of a system, that would, for any problem, automatically construct a working program from a description in a very high level language. This work has not yet produced a practical system that is easy for non-programmers to use; the difficulties in resolving ambiguities and inconsistencies in the problem statement seem intractable, in at least the near future. A second approach, that is practical, is to work within a limited problem domain where the problem is well defined and there is an available notation to resolve any ambiguities or inconsistences. These systems are call program generators. As an example, the program synthesizer used by many industrial corporations gives the capability to generate any of a whole class of similar programs and the user needs only to input special information related to his particular application. On the basis of this input, the system outputs reasonably standard code adapted appropriately for the

user's task. Examples of this product include program generators for industrial applications such as scheduling, inventory management, or payroll.

To put the user in a position where he or she need not be experienced programmers, the designer of a high level language should further simplify the so called "high level language" (e.g., FORTRAN). One method of simplifying a FORTRAN like language is by the collapsing of several lines of common patterns, such as the DO-loop or FOR-loop, into just one or two symbols. The language APL by Iverson (1972) [Ref. 17], gives an example of how this can be done. While the level of the APL language may not express the level to which the program generator is proposed, it does give a conceptual view of what is expected of the generator.

In accordance with the hierarchical model proposed for the Program Library, the program generator should also be represented as a level of the hierarchy. The level should be referred to as the application generator as shown in Figure 5.1. As with the Program Library, it should include various program generators consistent with the organizations overall goals (i.e., business, statistical analysis, etc.) The program generators should respond to the Application Generator's environment in a similar fashion to the way the libraries respond within the Program Library's environment. Therefore the program generator can be modified, and reused in a manner similar to that of a routine. As the figure implies the Inventory Management element of the application generator, being a program generator itself, must be viewed as being on the same level as all the other generators. Thus, each must be capable of communicating down to the various levels of the Program Library. An important restriction on the program generator is that its components are not permitted to communicate directly with each other.

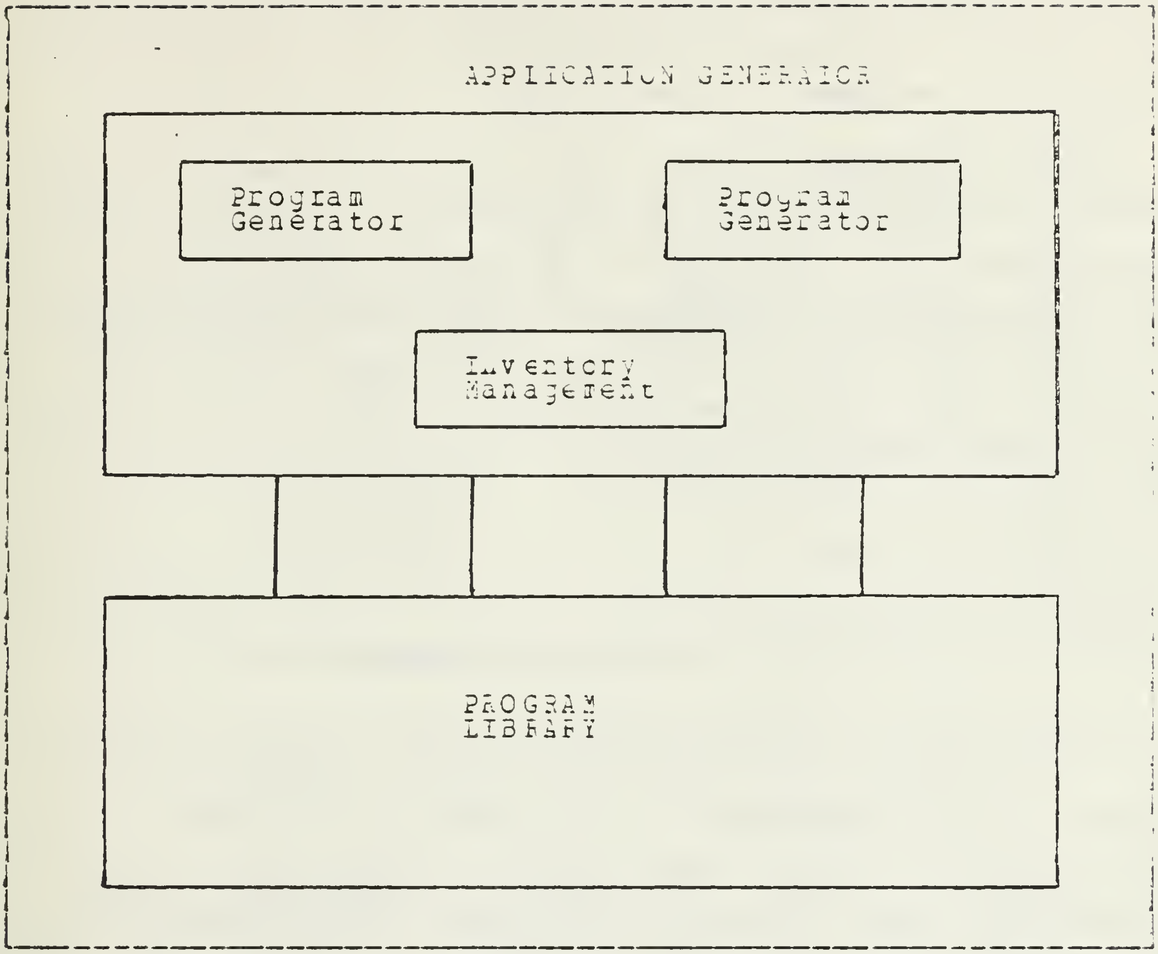

Figure 5.1 Hierarchical Structure with Additional application generator.

With the very high level application generator the hierarchical structure, previously presented, remains valid and now the user has a more user-friendly system.

A closer look at the application generator will illustrate one method of writing similar programs. The method is to segment the required task into two parts, routine portions that are common to all programs at that level and task-dependent portions that must be different for each new program. The program generator will respond as a program, that automatically executes the more routine portions of the program task and enables the user conveniently to input the

task-dependent information so that the desired program can be created. More detailed discussions on how this is accomplished can be found in [Ref. 18].

The simplicity of this software product becomes evident when the generator acts as an automatic program generator for applications specific to a working environment. The Program Generator's efforts are aimed at giving the user with no traditional programming expertise the ability to generate useful programs while working with familiar terms. The Business Definition Language (BDL) system being developed at IBM (Goldberg, 1975); Hammer et al, 1974 and PROTOSYSTEM I (Martin et al., 1974) at MIT are examples of an automatic program generator for the user's environment [Ref. 19], [Ref. 20] and [Ref. 21] respectively.

To provide a working example of the program generator, as it interacts with the user and the Program Library, Figure 5.2 is provided. The figure illustrates the flow chart created by the user within an interactive graphics program package. It also illustrates the interface between the generator and the Program Library. This interface is transparent to the inexperienced user but the experienced user is allowed access whenever he or she desires. The diagrams as shown describe the program as it is being created; they could also be thought of as the program or at least part of it. Since this generator can be described in terms of another such flow chart, then from a conceptual standpoint, more than one generator may be permitted in the application generator at the higher level. The generator at this point is still consistent with the hierarchical structure representing the Program Library and the environment surrounding it. The specific design of the program generator is to make the user's task as simple as possible. With this in mind, the following process, in Figure 5.2, is outlined.

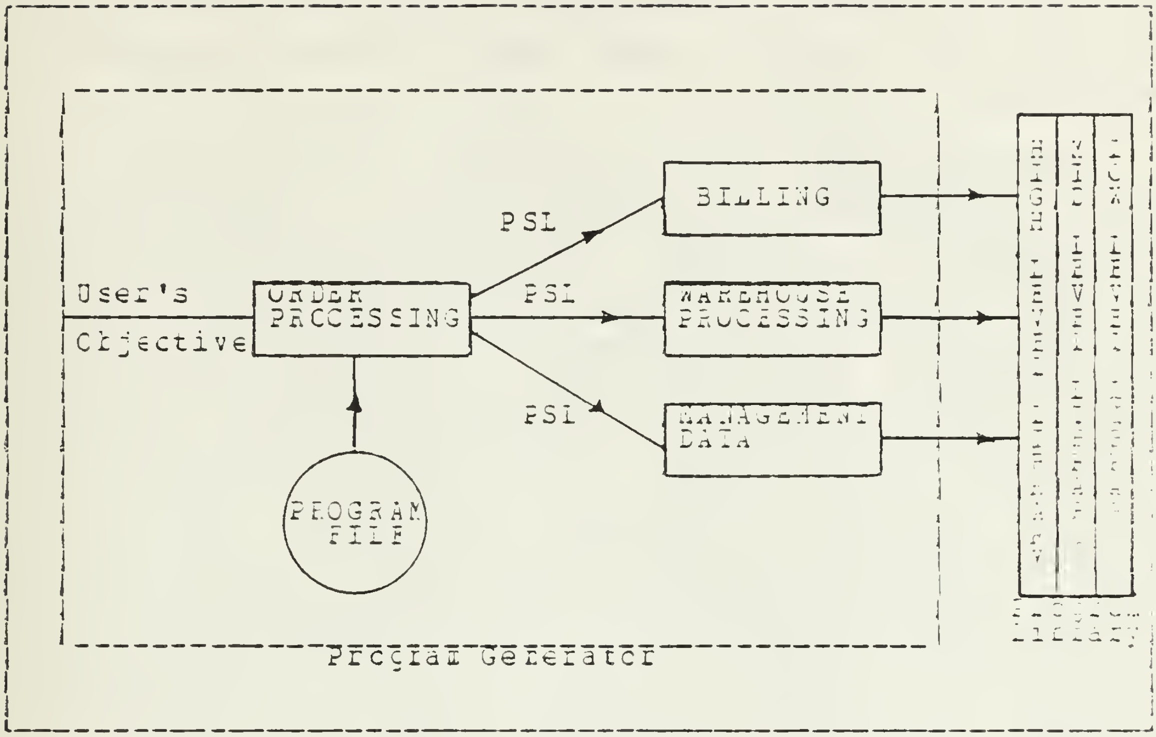

Figure 5.2 Example of a Program Generator.

Given a programmer concerned with processing orders, he must first make his objective clear to the interactive program package, presumably being used as the peripheral device. Incidental to the processing of orders, a check of the program file is made. This is to verify the existence of the required program, but not halt the process if the program is not present. Once the objective has been identified, the specific process of interest to the user is invoked by the Program Specification Language (PSL). The mechanics of the above operation is automatic in nature and transparent to the user. But, should the user require a more optimal solution, he or she has the capability of manipulating appropriate application packages or routines within the Program Library. The interface between the Program Generator and the Program Library must be well

defined and capable of extending the user's manipulative control down to the lowest level of the hierarchical structure.

The significance of this product is to introduce the notion of simplicity and of reusability. By implementing user-friendly (high-level) languages with special (task oriented) operators and forms designed for particular types of computation, repetitious coding of programs may be eliminated. These higher level languages are meant to include constructs that are adapted for particular applications and that are natural for conceptualizations in the problem domain. Hopefully, such languages will allow the programs to be concise and efficient. Since the generator operates somewhat automatically, the user's ability to produce correct and reliable programs should be considerably improved. This insures increased program productivity.

Examples of Program Generators and Program Libraries are in existence and marketable today, but as far as it can be determined, there is not a software product on the market that provides a combined environment, as exhibited above. This is not to imply that similar products do not exist. As an example of one organization's efforts at bringing the concepts of the generator and the library together, the following is worth mentioning.

The International Mathematical and Statistical Libraries, Inc. (IMSL) known for its numerical computational library, designed a system for a user so that:

(a) his programming effort could be reduced;
(b) he could have improved error control;
(c) he could have a system which is designed for ease of use, with the intent of increasing problem-solving productivity; and
(d) he would not be restricted to a single computer environment.

IMSL's solution to assisting the user is called "PROTRAN," (for PROgram TRANslator) [Ref. 22]. PROTRAN is designed as a family of software products, built around a preprocessor that produces FORTRAN code which performs the actions specified by the user's PROTRAN statements. The FORTRAN, thus produced by the preprocessor is combined with any FORTRAN the user may have written, and then it is compiled, linked, and executed. In order to reuse the PROTRAN programs on different problems, it is necessary to write the program in such a way that new data can be input and to insure that the command file (JCL, macro, etc.) does not delete the executable program file. The coordination between PROTRAN and the IMSL Library offers many advantages to the user, both which are highly sought in the Program Library and application generator.

1. Advantages of PROTRAN

The advantages of PROTRAN are extensive, so the following suggest only a few of the more dominate issues:

- Formal programming knowledge is not required for applications that can be done using PROTRAN statements alone.
- FORTRAN can be easily intermixed with PROTRAN statements, allowing a tailored approach to problem solving, for the benefit of the experienced user.
- Based on proven algorithms from the IMSL Library, it provides users with tested, reliable methods for problem solving.
- PROTRAN is powerful, flexible and ease to use. It has accurate and informative error messages and it allows unrestricted access to FORTRAN for specialized local requirements.
- It allows user to specify a programming problem in alternate ways.

- User documentation in machine readable form is made available to the system implementers. This allows them to generate a 'Help' facility for their users.

2. Summary

The intent of this section is to emphasize that there is a marketable need for software products, such as the application generator. More importantly,when combined with a Program Library, it provides a more functional product for the user and his working environment. The IMSL library should be viewed as a product which provides some lessons to be learned.

VI. INDEXING AND RETRIEVAL FROM THE PROGRAM LIBRARY

The Program Library offers much to the user of a software product. But the significance of the library is negated, if the user can not access and retrieve entities much faster than the time required to write an equivalent program. This search and retrieval process must create a working environment conducive to both efficiency and productivity. The library should be designed so as to include entities (i.e., routine and programs) and other program tools which will alleviate the need for a user to continually rewrite programs for each new application. The effectiveness is exhibited by the user's familiarity with the entities that are available and how they are called. Thus, the goal is to provide a Program Library which serves its purpose best by giving the user a fast way to locate entities. The concepts mentioned are not new, they have been studied extensively by Melinda Thedens [Ref. 23], with results that could make the Program Library highly effective. Thedens' results provide a conceptual view consistent with the idea of a Program Library.

The Program Library has been designed to support a hierarchical structure consisting of multiple levels of libraries, each accessible by the user. The entities within the library are well documented in a descriptive manner. Thus, the documentation can be used to assist the user with issues of form, parameter passing techniques, error handling procedures and any other standard features pertinent to the library and its manipulation. These and other features must be maintained to make the library effective, but the efficiency of the Program Library is more dependent on the speed

with which the library entities can be searched out by the user. Thedens suggests that a software product in association with the Program Library be used to help the users access and retrieve the needed routines and to explain how they should be used.

A. LIBRARY REFERENCE GUIDE

[Ref. 23] introduces the concept of a Library Reference Guide. The Reference Guide could be an on-line query program, a traditional manual that each programmer can keep on his (her) desk, or a combination of both. For the purpose of this thesis, the on-line query program will be the type reference guide described. The Reference Guide should be viewed as a software product which provides an interface between the user and the Program Library.

The Reference Guide, like the Program Library, has taken some ideas from the organization of the traditional library. One feature in particular is in the organization and indexing which functions like a card catalog. The index should consist of keywords that are used when calling up a selection of on-line files. This should be easily related to a user who is familiar with such traditional indexing tools as the KWIC (key-word-in-context) which accompanies the IMSL library. The indexing of files makes the user's task of locating entities much easier than writing them, but for the user to make use of the Reference Guide, it must also be simple to use. To maintain a high degree of simplicity, the description of what the entities are designed for should be organized; the organization should be such that the descriptions are kept to a few lines or steps. By maintaining short descriptions, the user is not bogged down with massive amounts of information which lessens the

degree of understanding while increasing the user's feeling of complexity. The short descriptions can be treated as well defined modules which can be modified to describe the entities that have also been modified.

With the documentation playing such a major role in the effectiveness of the Program Library, some of the concerns observed in [Ref. 24] should be reiterated. One concern is that the functional descriptions of how an entity performs its function internally should be hidden, so as to allow flexibility in writing future versions of the entities. The documentation should also contain a description of the inputs and outputs, particularly the formats and ranges of values. Finally, the documentation should include descriptions of the side effects of using the entities (e.g., which registers get destroyed, which work fields are used and which status flags are affected). The user should be able to use these items of information to avoid having to examine the code that performs the function.

The Library Reference Guide should be task oriented and the techniques of stepwise refinement should be used to describe the entities (from the most general to the more detailed levels). The importance placed on testing the entities of a Program Library should be extended to the documentation used to describe the library guide. The accuracy of the library documentation could be a deciding point as to whether the library's resources are used. The actual testing should involve checking for omitted information, information present in the wrong order, typographical errors, and ambiguous descriptions. Each time there is an update, or new addition to the guide, the above mentioned tests should be accomplished. The dates of these modifications should also be kept on file, so as to assist the user in identifying the changes as related to his particular

task. Therefore, the user will not be required to read the entire library, just that in his or her area of concern.

1. On-line Query Program

One approach to Thedens' on-line query program is to have it perform search and retrieve processes on the Program Library with the use of keywords. An example shown in Figure 6.1 can be described as follows: from the perspective of the user who requires a routine, but is unfamiliar with the specifics of the routine (i.e., what it is designed to do, what are its parameters, which routines does it call, etc.), a keyword or list of keywords can be extracted.

User's Query

The user can then establish a query from the identified (user's best selection) keywords. The user's query can be organized using different methods. One method consistent with [Ref. 23], suggests that every routine in the Program Library be described in short sentences containing a subject, a verb, and possibly a modifier. The words in the sentence which are not keywords (e.g., and, or, for, a, the, etc.) will be deselected by the translator. Another method is to provide keywords with boolean connectives; for example, given three keywords (A, B, C), they can be processed by the translator as A or (B and C). A scan of the library file would identify either keyword A or else both keywords B and C. A more likely strategy uses inverted indexes which, for each of the three keywords, contain lists of the document references exhibiting the particular keyword. The search process for the query then performs an intersection of the document reference lists corresponding to index terms B and C to identify items appearing on both lists. The resulting list is then merged with the document

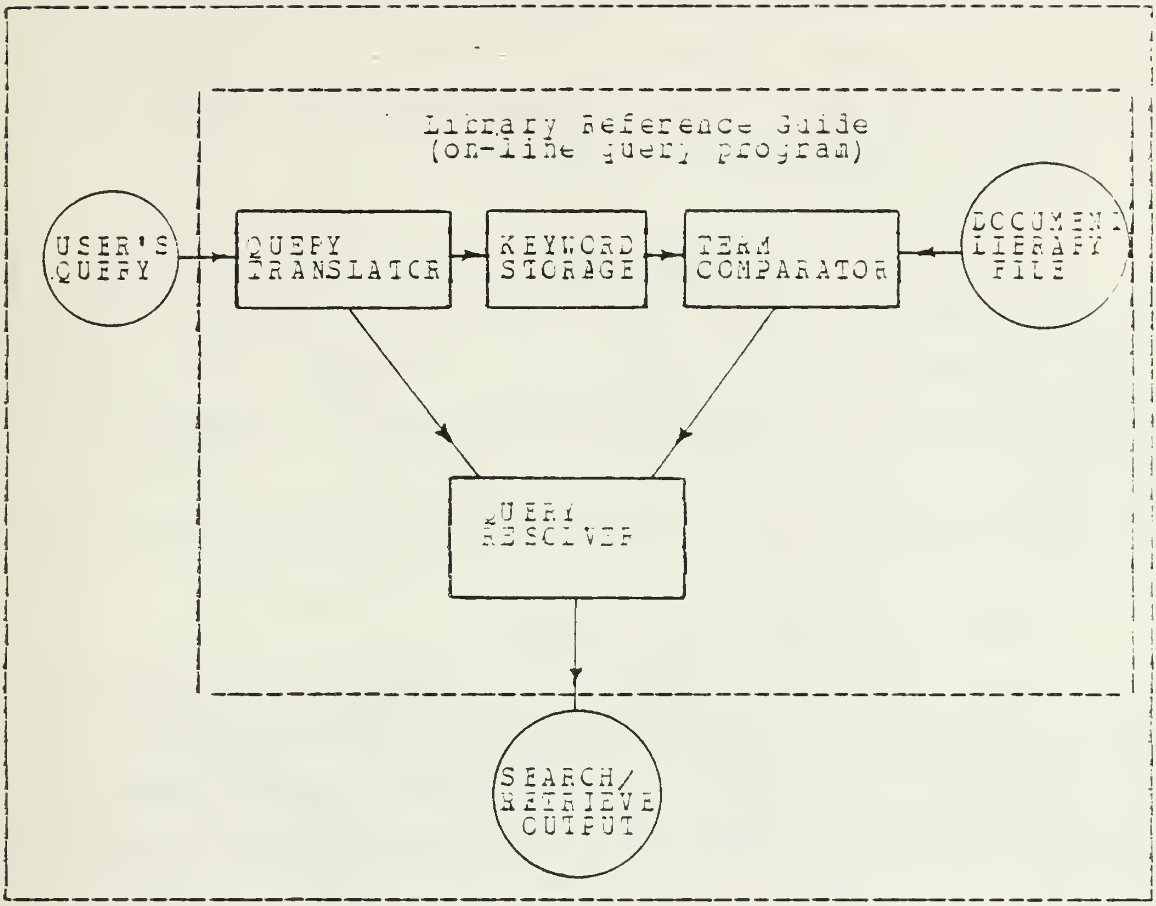

Figure 6.1 Matching of User's Queries Against Program Library Entities.

reference list corresponding to term A to obtain all items located either on the A list or on the combined B and C list. Independent of the method used, the translator will be required to handle the query. The latter method represents a quick search facility, thus it will be used to further explain Figure 6.1.

Query Translator

The query translator's function is to format keywords (i.e., break the query down into its component parts, (individual terms and boolean connectives)) for input to a temporary storage (e.g., a memory). The translator

must also maintain the query intact, so that it can be used later as a check against the routines and the keywords returned to the user. The keywords are maintained to allow the resolver to perform its functions.

Keyword Storage

The keyword storage acts as a memory for storing distinct terms (keywords) temporarily in a predefined format (i.e., parallel with n cells for n terms). The keywords should be held in storage until the search process has been completed or until deselected by the user. The format of the terms is important to the next step of the process which uses the term comparator.

Term Comparator and Document

The comparator matches the identifying information from the document library file against the query terms. To avoid having to page through the entire library file, the comparator receives only the keywords associated with the routine's function. The comparator should be built to handle truncated terms (with missing prefixes as well as missing suffixes and so-called "don't care" characters). With this facility the question of ambiguity must be addressed. Figure 6.2 shows a hierarchy of keywords, associated with similar, but different routines. The ambiguity becomes a factor when the routines are searched using the truncated keyword, thus calling the routine INIT or INIT* could return either of the structures. To avoid ambiguity the comparator will return both routines, giving the resolver or eventually the user, the option of selecting the appropriate routine. The terms returned to the comparator are possible because, as Thedens suggests, the library guide is constructed such that each entity (i.e., program, routine, etc.) is preceded by documentation information consistent with a design template. The template will be

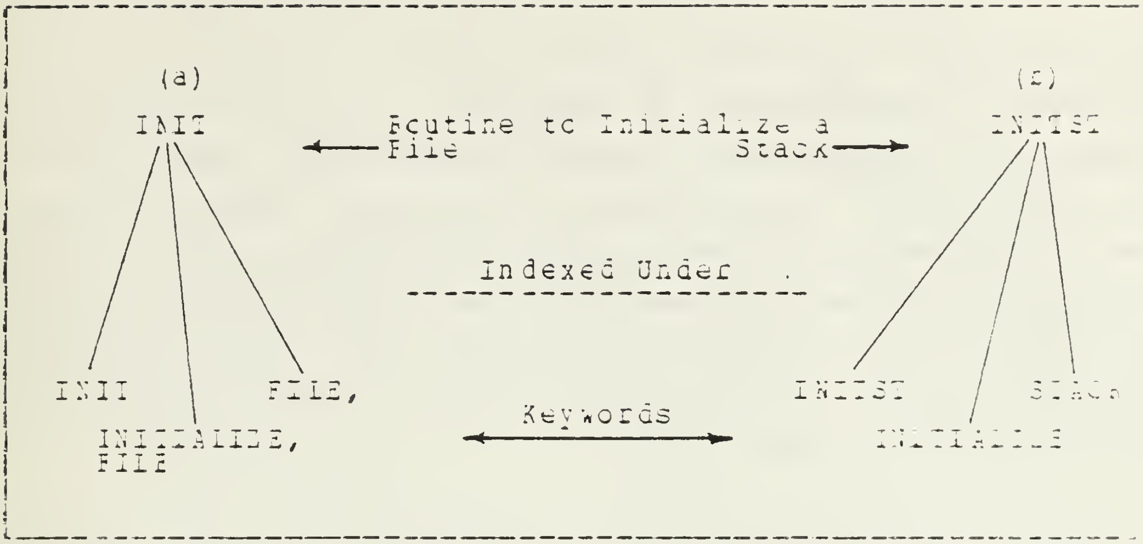

Figure 6.2   Hierarchy of Keywords.

designed with a consistent format so as to allow the query program or the programmer when required, to select the information needed without scrolling the entire routine. An example of possible template heading might include:

Description
Keyword
Format
Originator/Project
On Inputs
On Return
On Error
On Calls
Requirement
Options
Special Case
Examples
Updated
Found In
See Also
Uses

Since the template is standard there will be some routines which will not use all of the headings, the ones that don't apply should be deleted. With the "uses" and "found in" heading the search process can take on the appearance of a hierarchical structure. This approach, shown in Figure 6.3, can easily be adapted for use with the hierarchical structure used in the Program Library (i.e., the top of the documentation should point in the right direction and the search of subsequent lower layers should provide more and more detail).

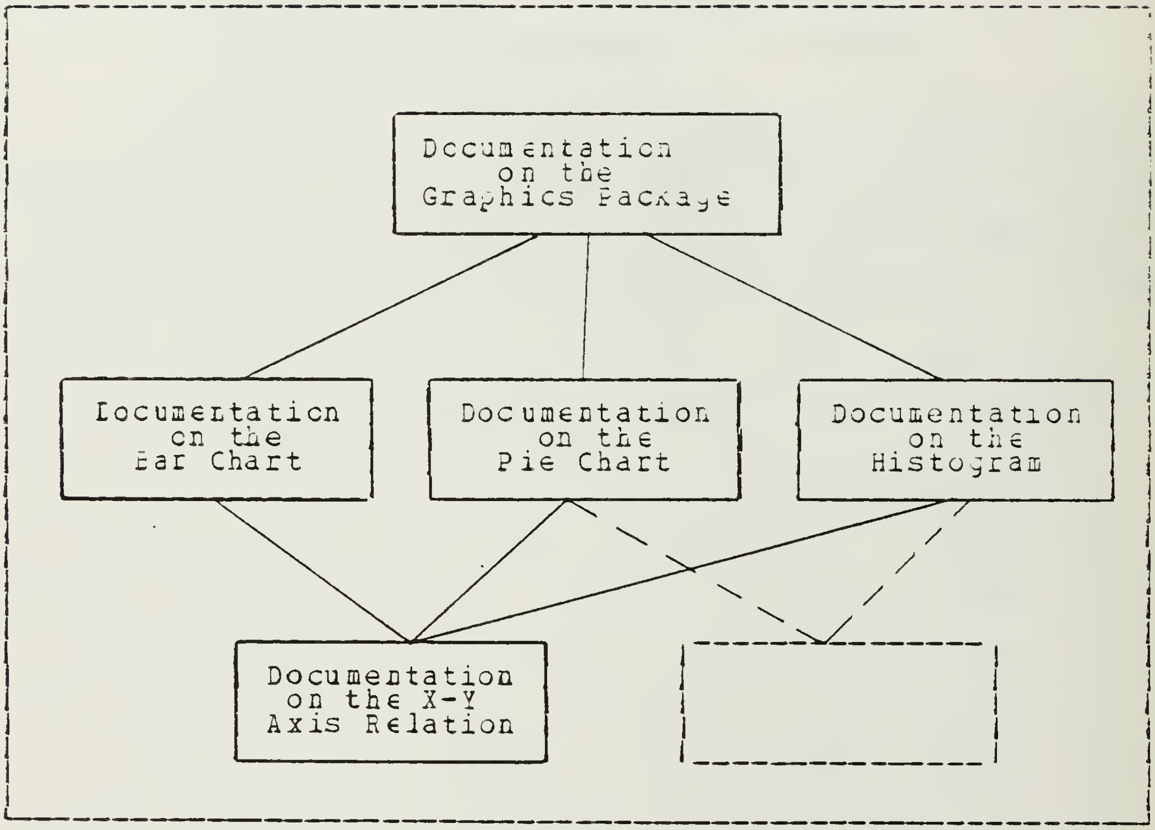

Figure 6.3 Hierarchy of Reference Guide Documentation.

Query Resolver

The query resolver checks whether the complete user query statement is fulfilled by the matching document library terms. Should a returned routine not be consistent with the user's query, it is deselected. This means the user will not necessarily see all the routines selected by the comparator. Once the checking stage has been completed, the results are sent to the output device (e.g., a terminal or a line printer).

Search Output

The actual output will consist of a list of routines by name, with a short descriptive abstract of the routine's function and other related keywords. Since the document library file only contains the header template rather than the documentation plus code, the user should be able to view the remaining documentation of the routine. This could be compared to a "browsing" facility which provides feedback for query refinement.

Retrieval

Once the user has located the desired routine name and is confident that it does the required task, he or she can tag the routine for retrieval at the end of the browsing session or initiate the retrieval at that time. When the routine has been tagged for retrieval, its location within the Program Library is identified (i.e., pointer directs the system to its location in memory). The user can now be prompted as to whether the routine is to be retrieved (i.e., placed in the user's file). At this point the retrieval process will permit authorized users to continue the browsing process down to the actual source code level, it will also allow updating (i.e., additions and deletions) and most any manipulation permitted in the program library. The retrieval is similar to the search process in that it

depends heavily on the names and functions of the entities in the program library. A major distinction between the two processes is the presumption that when the user invokes the retrieval process he or she knows the identity of the entity and is fairly sure of its location in the program library.

Prior to using the retrieval mechanism the user should be familiar with the hierarchical structure of the Program Library. A simplified example of what the user should envision in the structure and what procedure could be used to retrieve a routine at different levels will be presented. First, the user should have a general understanding of the library's structure for a specific implementation. The following should provide a general assessment of the structure. The structure can be viewed as containing entities which are refered to as its members. The members of the structure are ordered hierarchically. The main members at the higher levels of the library are called supersets to any member at a lower level and likewise any member at the lower level is called a subset of the higher levels.

A structure should define its organization and the names of the members on each level in the structure. A general form of the structure could include:

- the name of the member at the highest level
- the names and attributes of its members and
- a level identifier for each name to define its level in hierarchical order.

Examples of this structural form can be seen in the record structure of a PASCAL program or the structure declaration of a PL/I program. To illustrate what the user could expect when retrieving a routine from the Program Library the following scenario is proposed.

To begin the scenario, a programmer is given the facilities (i.e., hardware and software) of a potential workstation. He or she is expected to take these facilities and establish a workstation capable of assisting in accomplishing their task or job. One major asset included in the facilities is a Program Library. The library contains the routines to build essential programs. These programs consist of the appropriate routines to make a workstation responsive to the programmer's requirements.

The facilities provided to the programmer are similar to those of a SUN workstation and thus include: the capability to operate in a batch or an interactive environment, the ability to use various input devices (e.g., the joystick, the mouse, the track ball and the touch screen), the ability to produce either color or monochrome displays (with varying hues and x-y addressing) and the facility to respond to a number of different software packages (e.g., DBMS, Graphics, Games and Inventory Management).

The programmer must now establish the correct facilities to allow the workstation the capable of performing the desired task. The necessary data can be retrieved from the Program Library as shown in the hierarchical structure of Figure 6.4. In the example, the programmer requires a graphics package, which is user interactive, with a color display controllable with a mouse. The routines which will give these and other features are stored in the Program Library until retrieved by the programmer for insertion into a program.

Figure 6.5 uses Jot notation to illustrate how the programmer can retrieve routines from the Program Library. With the use of a library prompt (Library >) the various

routines can be located and retrieved as shown. The dashed lines around the routines imply that the programmer should be able to retrieve a routine directly without going through its immediate superset. For example, Library > Lib.Graphics.Moveto can be used to retrieve the low level routine Moveto, without using routines at the Mid levels.

Ambiguities can arise when referencing the members of a structure because the name of a member can occur as the name to more than one superset. To resolve such ambiguities, qualified names to reference members of the library structure, can be used. In a qualified name, the member name is preceded by a list of routine names in appropriate order by levels, each followed by a period. The only routine names required are those that determine a unique reference to the member name. For example, in the following structure

```
1 Graphic
  2 Interactive
    3 Color
    3 Mouse
      4 Moveto
      4 Lineto
      4 Drawtext
  2 Batch
    3 Color
      4 Display
        5 Moveto
        5 Lineto
        5 Drawtext
```

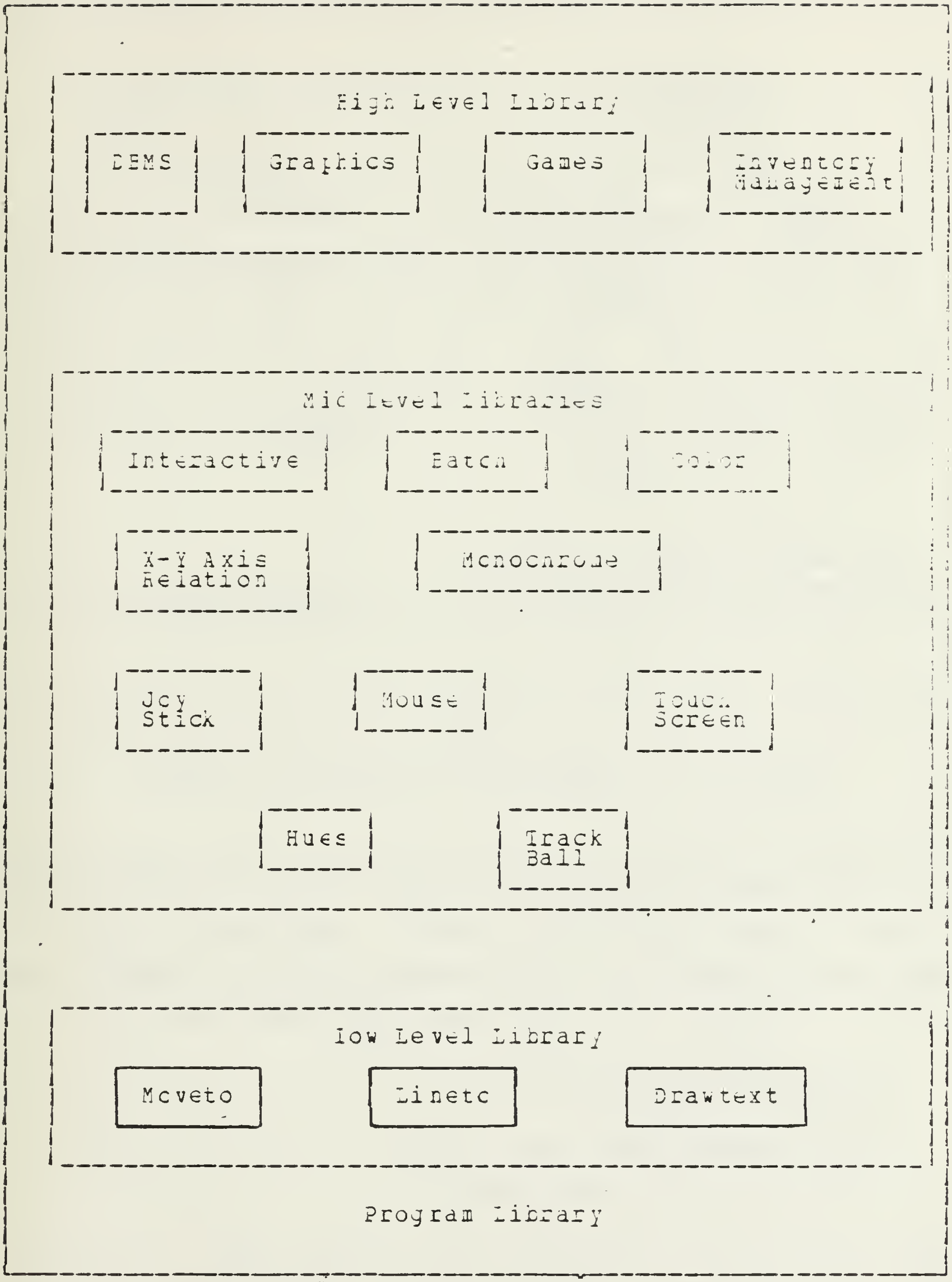

Figure 6.4 Example Hierarchy of a possible Program Library.

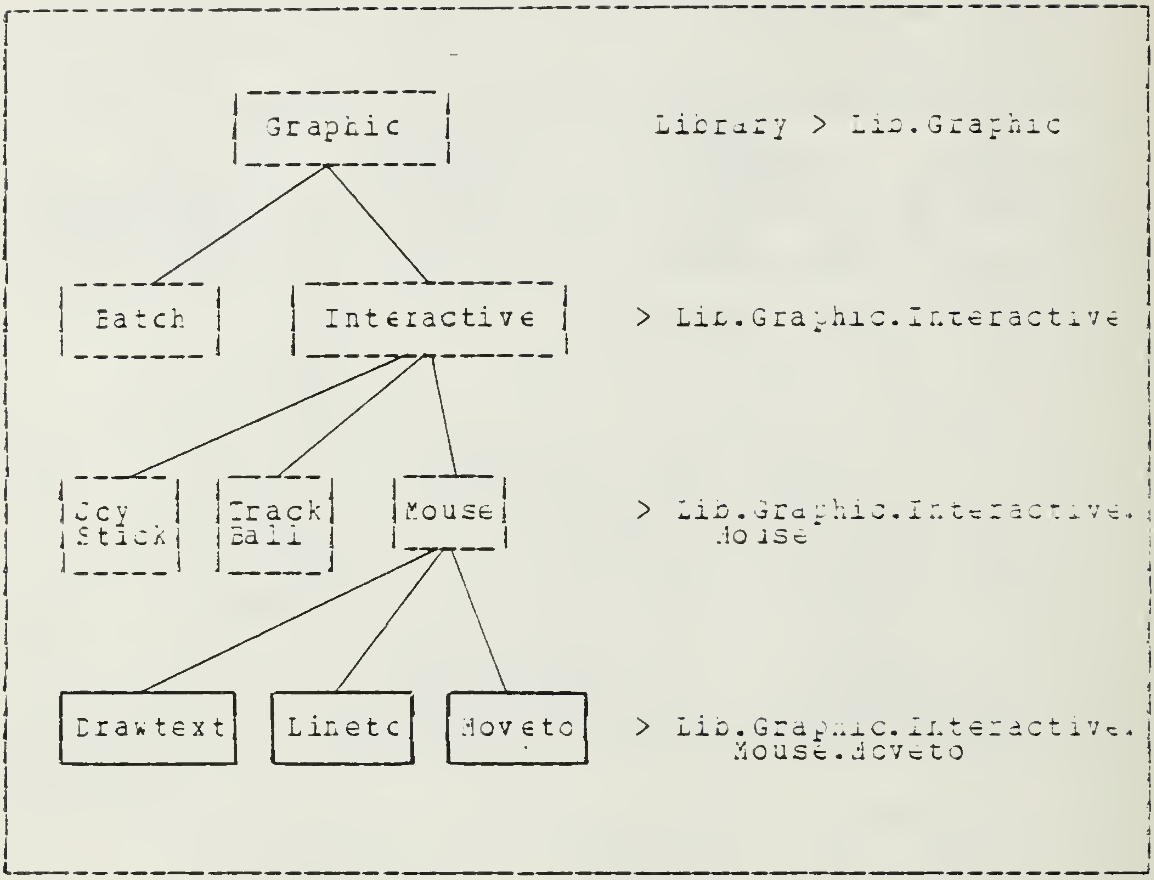

Figure 6.5 Example of a Retrieval Process.

a reference to Moveto, Lineto or Drawtext, or Graphic.Color or Graphic.Moveto is ambiguous along with a few other relations. The qualified names Interactive.Color or Interactive.Moveto, or Batch.Moveto uniquely identify the library routines. The fully qualified names would be

Graphic.Interactive.Color

Graphic.Batch.Color

Graphic.Interactive.Mouse.Moveto

Graphic.Batch.Display.Moveto

each should help to alleviate any ambiguities. To shorten the user's request truncated names can be used, but as

illustrated previously, the other forms of ambiguity must be resolved.

E. SUMMARY

The anticipated goal of the Library Reference Guide is to simplify the search and retrieval from and additions to the Program Library. Simplicity of both issues should improve on the user's efficiency, lessen the amount of time wasted looking for the best entity, improve the users ability to write programs and finally and most important, increase the user's productivity.

VII. THE ADA PROGRAMMING LANGUAGE AND THE PROGRAM LIBRARY

Goguen [Ref. 25], discusses the cost of software tools: "It appears that each successive generation of software development tools has been significantly more expensive than the previous one. However, these tools are still much less expensive than corresponding hardware tools, such as fabrication lines." Even with this knowledge there is still great reluctance to invest significant amounts of money into research and development for software tools. In fact, since it has been discovered that most of the cost of real systems now lies in software rather than hardware, the reluctance to invest becomes even more evident. There are some hopeful signs which have shown that the Japanese "software factories" are actually capable of achieving rates of reusability ranging from 60% to 80% [Ref. 25]. Also, some U. S. industries and specifically the Department of Defense (DoD) have begun to invest in the field of software productivity. DoD's efforts have been extensively geared to the development of the programming language "Ada," which is designed in accordance with requirements established by the DoD.

The requirements call for a language with considerable expressive power covering a wide application domain. As a result, the language includes facilities offered by classical languages such as Pascal as well as facilities often found only in specialized languages. Thus, the language is a modern algorithmic language with usual control structures and with the ability to define types and subprograms. It also serves the need of modularity, whereby data, types, and subprograms can be "packaged."

The four program units of Ada are subprograms, package units, task units, and generic units. The two units of special interest for a Software Library are the package unit and the generic unit. A package is defined as a collection of logically related entities. A generic unit is a template, which is parameterized or not. Corresponding (nongeneric) subprograms or packages can be obtained from them. The resulting program units are instances of the original generic unit and thus, forms of "instantiation."

An example of how Ada supports the Program Library is in the way the generic program unit can be used. It has been suggested that each entity in the Program Library contain a heading template to be used as a means of searching and retrieving the entities for possible modification (i.e., deletion, addition and updating). What the generic program unit provides is the ability to not only search and retrieve, but also to minimize the modification. One method in which Ada exhibits this is shown below [Ref. 2], where a subprogram is created that exchanges two elements of an integer type:

```
procedure INTEGER_EXCHANGE(FIRST, SECOND: in out INTEGER)is
   TEMPORARY : INTEGER;
begin
   TEMPORARY := FIRST;
   FIRST     := SECOND;
   SECOND    := TEMPORARY;
end INTEGER_EXCHANGE;
```

Once this application is established other types of elements may be exchanged without creating a new subprogram for each instance. With the algorithm being identical in all cases, the similar operations may be factored out by adding the following generic unit to the procedure specification:

```
generic
   type ELEMENT is private;
procedure EXCHANGE (FIRST, SECOND : in out ELEMENT);
```

The body now becomes:

```
procedure EXCHANGE (FIRST, SECOND : in out ELEMENT) is
   TEMPORARY : ELEMENT;
begin
   TEMPORARY := FIRST;
   FIRST     := SECOND;
   SECOND    := TEMPORARY;
end EXCHANGE;
```

The significant portion of this subprogram specification is the addition of a prefix, called the "generic part," that defines all of the generic parameters (if any). The above two algorithms have the same identical body with the exception of the data type which is handled by the generic part. This process, as shown, allows the programmer the ability to make use of the existing body of a program unit, instead of writing one from scratch. So with this method, the modifications are mainly performed on the specification (i.e., the generic part), hopefully minimizing the degree of change necessary. The Program Library would manipulate its entities in a similar manner, making it at least as reusable as Ada makes its generic packages.

Since generic units are just templates, they are not executable, and so they may not be used directly. But they create instances of the generic unit. Thus, the instantiation of the generic unit makes the subprogram or the package sufficiently easy to identify and combine with other units. Therefore, the goals and concepts of the Program Library are supported by the Ada program language and although Ada may

not represent the best language for the library, it does support the Program Library concept, see [Ref. 25].

With [Ref. 25] and [Ref. 2], the aforementioned examples are made clear and the Program Library is established as a potentially feasible software product. Even more supportive is the reference made to the organization of a "Ada Program Library." The reference makes similar proposals to those of this thesis in the area of library construction and operation. Specifically it suggests a hierarchical classification scheme, with different levels of detail and formalisms, and with each entity accessible by keywords. This does not imply that the proposals offered are all inclusive or anywhere near ready for implementation. However, the concepts are not that remote and at least one organization (i.e., the DoD) is willing to risk the time and money to investigate the potential to achieve these conceptual goals.

VIII. NON CODE PRODUCTS IN SOFTWARE LIBRARIES

Even with the Software Library representing an effective reusable software product, one must ask if that is enough to encourage effective software development. The Software Library has been represented by products (e.g., the Program Library) designed to enhance reusability of code. Although the management and organization of code is critical to the future development of reusable software, there are other software products that are developed during the life-cycle that have the potential for reuse. These include documents, requirements, specifications, designs and test plans. Just as reusability in coding can be used to reduce software coding costs, so can reusability of software products in other phases of the life-cycle contribute to cost reductions. Each of the concepts in the conceptual Program Library can be applied to other software products in the life-cycle.

The definition of reusability has placed the emphasis on the capital returns of a software product. If it is more cost effective to use existing designs, specifications, requirements and test plans, then they should be reused. Even with this being the case, if they are not organized in an accessible and retrievable manner, they lose their reusable nature. With reusability being so important, this issue must be addressed as an objective of the development process throughout the life-cycle. To reduce the overall cost of a software product, all phases of its life-cycle should incorporate methods and standards which will support reusability. Once the software product is postulated as being reusable, the issue must be fully addressed then and

not at some later phase of the life-cycle. However, the issue of reusability should not be forced on the design, specification or any other phase not compatible to the required application. Each phase should be viewed separately and a determination made as to whether reusability is economically feasible. If it is then reusability should be incorporated, but if it is not feasible it should not be insisted.

Finally, since reusable software products offer an approach to lessening the effects of the "Software Crisis," environments encompassing this concept should be established. These environments should be concerned with phases of the life-cycle other than code. That which has already been learned from working with reusable code should be applied, thus avoiding the "reinventing of the wheel."

IX. CONCLUSION

The "software crisis" is real and if the computer industry is to have any impact on reducing its effects, software developers must begin making concerted efforts to create reusable software products. This thesis has presented the Software Library and its prototype the Program Library as possible reusable software products. Methods of making the concept of a Software Library better understood by the user were discussed. This was accomplished by comparison of the Software Library to a traditional library and by relating it to other program libraries (particularly the IMSL and the NAG). These comparisons yielded characteristics which could be associated to a quality Software Library.

If the Program Library has a hierarchical structure, then the entities within the library can be easily accessed and retrieved by a user. Reusability is thus established as a viable solution to some of the economic problems in software development.

Application generators with similar hierarchical structure to the Program Library can be used to assist the inexperienced user perform his or her task. The experienced user should be allowed to modify entities in both the Program Library and the application generator.

An on-line query program was discussed as an interface between the Program Library and the user. The query program is one approach to bringing reusability to the software product.

The Ada programming language was described as having features that support the concept of a reusable Program Library. Future concepts in the Ada program library which are in line with the issues in this thesis are referenced.

Finally, the Software Library should include products from phases of the life-cycle other than coding. Documentation, specifications, requirements, designs and test plans should be incorporated into the concept of a Software Library.

LIST OF REFERENCES

1. Ellis Horowitz and John B. Munson, "An Expansive View of Reusable Software," *Proceedings - Workshop on Reusability in Programming*, pp. 250, 1983.

2. Grady Booch, United States Air Force Academy, *Software Engineering with Ada*, The Benjamin / Cummings Publishing Company, Inc., Menlo Park, Ca., 1983.

3. Peter Wegner, "Varieties of Reusability," *Proceedings - Workshop on Reusability in Programming*, pp. [illegible], 1983.

4. John R. Rice and Herbert D. Schwetman, "Interface Issues in a Software Parts Technology," *Proceedings - Workshop on Reusability in Programming*, pp. [illegible], 1983.

5. Druffel, L.E. *Draft Software Initiative Plan*, DoD, DUSD(R&AT) CSS, October, 1982.

6. William C. Johnson, *Reusable Software - Assessment and Potential*, MS. Thesis, Naval Postgraduate School, Monterey, California, 1984.

7. John R. Rice, "Software for Numerical Computation," *Research Directions in Software Technology*, ed. Peter Wagner, MIT Press, January 1980.

8. Peter Freeman, "Software Reliability and Design," from Proceedings, 13th Design Automation Conference, June 1976, *Tutorial on Software Design Techniques*, Third Edition, edited by Peter Freeman and Anthony I. Wasserman, 1980.

9. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *Tutorial on Software Maintenance*," edited by Girish Parikh and Nicholas Zvegintzov, pp.135-145, 1983.

10. Don R. Swanson, "Design Requirements for a Future Library," *Libraries and Automation*, edited by Barbara Evans Markuson, Library of Congress, Washington, D.C., 1964.

11. T. J. Aird, E. L. Battiste and W. C. Gregory, "Portability of Mathematical Software Coded in Fortran," *ACM Trans. Math. Software*, 3, pp. 113-127, 1977.

12. *IMSL Library Reference Manual*, edition 9, by IMSL, Inc., 1982.

13. B. Ford and D. K. Sayers, "Developing a Single Numerical Algorithms Library for Different Machine Ranges," ACM Trans. Math. Software, 2, pp. 115-131, 1976.

14. J. A. Prentice, "The Development and Maintenance of Multi-Machine Software in the NAG Project," Software for Numerical Mathematics, (D. J. Evans, ed.), Academic Press, London, pp. 383-392, 1974.

15. S. J. Hague and B. Ford, "Portability-Prediction and Correction," Software Practices and Experiences, 6, pp.61-64, 1976.

16. R. M. Balzer, Imprecise Program Specification, USC-ISI, No. ISI/RR-75-36, December 1975.

17. K. E. Iverson, A Programming Language, Wiley, New York, 1972.

18. Alan W. Bierman, Approaches to Automatic Programming, Computer Science Department, Duke University, Durham, North Carolina, 1976.

19. P. C. Goldberg, "Automatic Programming" Programming Methodology, (G. Goos and J. Hartmanis, eds.), Lect. Notes Computer Science, Vol.23, p.347, Springer-Verlag, Berlin and New York, 1975.

20. M. M. Hammer, W. G. Howe, and I. Wladawsky, "An Interactive Business Definition System," Proc. Symp. Very High Level Language, pp.25-33, 1974.

21. Massachusetts Institute of Technology, Cambridge, W. A. Martin, M. J. Genberg, R. Krumland, B. Mark, M. Margenstern, B. Niamir, and A. Sunguroff, Internal Memos, Automatic Programming Group, 1974.

22. IMSL's User News on "PROTRAN," IMSL Inc., Houston, Texas, March 1983, June 1983, and September 1983.

23. Melinda Thedens, Cataloging the Program Library, Currently a communicator for ATEX Inc., May 1983.

24. Thomas Fay, Subroutine Libraries, A senior applications programmer at National Semiconductor Corp., Santa Clara, Ca., May 1978.

25. Joseph A. Goguen, Suggestions for Using and Organizing Libraries for Ada Program Development, pp. 32-33, SRI International, Menlo Park, Ca., 1983.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Commander Naval Data Automation Command Washington Navy Yard Washington, D.C. 20374	1
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	Professor Norman R. Lyons, Code 54Lb Department of Administrative Science Naval Postgraduate School Monterey, California 93943	1
6.	Professor Gordon H. Bradley, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
7.	Commanding Officer Attn: Lt Sherman G. Metcalf Integrated Combat Systems Test Facility San Diego, California 92152	3
8.	Computer Technology Curriculum, Code 37 Naval Postgraduate School Monterey, California 93943	1